Developing Critical Software in the Modern Threat Environment

Brian Stevens[A], Rob Ashmore[A], Andrea Margheri[B] and Vladimiro Sassone[B]

[A] Defence Science and Technology Laboratory (Dstl), Portsdown West, UK

[B] Electronics and Computer Science, University of Southampton

Abstract *As software becomes ever more embedded into the fabric of society, more systems are becoming critical to large numbers of people, either by design or unintentionally. Even those that may not be considered safety-critical can have a large impact when they fail (e.g. banking systems). Consequently, software can be critical for a number of reasons, including: safety; security; and mission impact of failure. We would expect criticality, along with software requirements, to emerge from coherent, integrated systems-level analyses that include data, security, mission and safety aspects. We have combined software requirements from a number of sources, including those based on the "4+1" software safety principles and those emerging from security considerations, to produce a single list of top-level expectations that any critical software development would be expected to satisfy. This list provides a simple, unified structure that may, for example, be used to organize audits or promote discussion between customer and supplier.*

1 Introduction

Software is becoming increasingly embedded into the fabric of our society and we are becoming critically reliant on it functioning correctly. This software ranges from large systems (*e.g.* air traffic management) to small, often forgotten, pieces of embedded software (*e.g.* drivers in a USB stick). Parts of this software may be important for reasons of safety, security or mission criticality and special care must be applied in the development of these items.

At the same time as our dependence on software is increasing, new threats are emerging. In earlier decades, software-related issues tended to occur because the environment had produced an unexpected input, which exposed behaviour not explicitly covered by a test case. Now, threats are posed by those actively wishing

to disrupt systems, for example, for monetary or political gain. The large variety of frameworks freely available for automated software vulnerability investigation has significantly increased the number of subjects potentially posing threats: off-the-shelf scripting security attacks can still pose critical threats and endanger critical safety and mission requirements. Given this situation, it is crucially important that software development processes address all types of criticality.

In this paper, we build on our software, safety and cyber security experience to propose a structured list of expectations supporting the development of modern software systems.

The remainder of this paper is structured as follows. In Section 2 we outline our scope and terminology. Section 3 provides background on critical software, whilst Section 4 highlights key attributes of the modern threat environment. Section 5 summarises our method. Our expectations are described in Section 6. Summary conclusions are provided in Section 7.

2 Scope and Terminology

Given the broad scope of software and development terms, we first formalise their use in our paper. Then, we introduce the primitive types of software component intended for our analysis.

2.1 Software

For the purposes of this paper, we use the term "software" to encompass a wide variety of programmable elements, including:

- Traditional Software: That is, software written in languages such as Ada, C++, Java and assembler. This is normally loaded from non-volatile storage when required and may be readily updated; typically, these updates do not require changes to the underlying computational hardware.
- Firmware: That is, software produced in the same manner as traditional software, but embedded within a device prior to delivery. Typically, changes to firmware are associated with changes (*e.g.* in part number) to the associated device.
- Complex Electronic Hardware (CEH): That is, software contained within devices such as Application Specific Integrated Circuits (ASICs) and Field Programmable Gate Arrays (FPGAs). Although there are some similarities, this software is developed using different approaches to traditional software and firmware.
- Algorithmic Data: That is, collections of data items that influence system behaviour, including configuration and adaption data. These are typically used to

increase software flexibility, in particular by giving a greater amount of control to an end user.

All of these types of software need to be developed appropriately and all are within the scope of this paper.

2.2 Development

For brevity, we use the term software development to cover all aspects of software: management; development (including design and implementation); verification; and maintenance. All of these activities are within the scope of this paper.

Although we recognise the importance of regular communication between software and system teams, activities of the latter are outside our scope. In particular, we expect system-level activities to have produced a set of structured requirements that are allocated to software. We also expect system-level activities to have assigned criticalities to these requirements (or the system functions they deliver) through the application of a rigorous, repeatable and auditable process.

2.3 Item

To simplify discussion, this paper often refers to a Computer Software Configuration Item (CSCI), which is, "An aggregation of software that is designated for Configuration Management (CM) and treated as a single entity in the CM process" (ISO, 2017).

In addition to that specific definition, we also use the term CSCI as convenient shorthand to refer to controlled software that is produced as part of the development process. In some cases the development process may produce a single CSCI; in others, a collection of related CSCIs may be produced from the same process (*e.g.* using the same planning documents, *etc.*).

3 Critical Software

Software can be critical for many reasons and be developed by many types of process. To proceed with our analysis, we first introduce the precise meaning of critical software and its potential hidden criticality factors.

3.1 Reasons and Levels

Historically, the notion of "critical software" has been inextricably linked with "safety-critical software". However, the way that software is now used and the diverse range of threats it faces mean we need to widen our understanding of what critical software means. We believe there are three main reasons a CSCI to be deemed critical:

- *Safety*, where software failures lead to loss of life, significant loss of equipment or significant environmental damage. Aircraft flight control software is an example.
- *Security*, where software failures adversely impact confidentiality, integrity or availability of information. Stock Exchange control software is an example.
- *Mission*, where software failures prevent the achievement of operational goals (*i.e.* failures stop the software delivering what it was designed to). Software that displays train departure information at a railway station is an example.

Although these categories are conceptually simple, it is difficult to precisely define a critical / non-critical boundary for any of them. For example, it could be argued that simply flying an aircraft causes *some* environmental damage; the point at which environmental damage becomes significant is more debatable.

This challenge can be eased by defining a range of *criticality levels*, for example, Safety Integrity Levels (SILs) in IEC-61508 (BSI, 2010) or Development Assurance Levels (DALs) in ARP4754A (SAE International, 2010). At first sight, providing a greater range of levels appears to complicate our problem of defining boundaries and increase the chance of a CSCI being misclassified. However, multiple levels allow us to adapt our development approach in a graduated manner, which reduces the consequence of a (minor) misclassification.

As indicated by the references above, allocation of criticality levels to software functions (and consequently to CSCIs) is best addressed at the system level. As such, it is outside the scope of this paper.

That said, the close relationship between accidents, system-level hazards, software functions and software criticality levels is obvious. Consequently, the lack of established guidelines on how to integrate safety, security and mission critical development standards may lead developers to inappropriate decisions which may be in conflict. By way of example, the same system can be designed differently by only considering safety or security standards. For example, consider a train's emergency brake system. From a safety perspective, the system may be designed to be as easy to use as possible; from a security perspective, two-factor authentication may be implemented, to protect against misuse.

In this context, we note that in System Theoretic Process Analysis (STPA) (Ishimatsu, 2014) accidents are things that a stakeholder would consider as a loss. Consequently, STPA provides a coherent framework within which all three reasons for software criticality can be considered together. Its flexibility allows

STPA to be easily tailored and extended in order to include new development principles, for example cyber security principles (Howard, 2017).

Being able to treat all three reasons for software criticality in a coherent manner alleviates potential difficulties associated with boundaries between them. As such, the expectations we propose in this paper try to harmonise the fundamental development principles of each of the reasons. This is helpful as establishing meaningful boundaries is likely to be impossible. For example, in the modern world, a system cannot be safe if it is not secure and safety is often a key part of the system's mission.

3.2 Hidden Criticality

The preceding discussion focussed on reasons why software may be critical from the perspective of the system it supports. This the most obvious way that software can become critical, but it is not the only way. For example:

- *Functional misuse.* The software may be correct with respect to its specification, but its behaviour may not be properly understood by the user. For example, there have been cases where patients have been given inappropriate doses of radiation during treatment (Ash, 2007). Potential causes include inadequate training and lack of documentation in the user's language (*i.e.* user-level documentation was critical).
- *Architectural misuse.* The wider system within which the software is implemented may be used inappropriately. For example, there are cases where blind obedience to a satellite navigation device has led car drivers into difficulties. This is a specific example of a more general problem: over-reliance on advisory items.
- *Escalation of non-critical functionality.* A piece of apparently low criticality software in one system may (unintentionally) allow the behaviour of high criticality software in another system to be altered. For example, an Internet-enabled fish tank was used to exfiltrate data from a North American casino (Matthews, 2017).

Given the almost infinite number of ways that modern software-bearing systems can interact, identifying all routes by which a CSCI may become critical seems an impossible task. Nevertheless, there are some simple, practical things that can be done, including: educating users on intended software behaviour and acceptable system use; driving up minimum software standards (*e.g.* by increasing the quality of supporting tools and frameworks); and providing strong partitioning of obviously critical software.

4 Modern Threat Environment

Nowadays, software systems need to face a large variety of threats, each of them of different impact and with different originating motivations. Threats are not caused just by software implementation errors, but also by vulnerabilities in CSCIs. A single vulnerability of a component can lead to severe threats for the associated system.

Differently from the past, the freely-available, off-the-shelf vulnerability assessment tools (Kali Linux, 2018) allow any programmer to potentially become a threat agent. As a matter of fact, scanning and searching for known vulnerable configurations or versions of software can be done by so-called "script kiddies". These are technologically-minded individuals that for curiosity, for fun, or for the desire to be a hacker, challenge themselves to succeed in a cyber-attack. Although surprising, this is still a significant threat that, despite the low attack difficulty, can cause severe harm.

More severe threats are posed by cybercriminals and hacktivists. Both groups represent highly skilled hackers and are the most active threat agent groups in the modern environment. Cybercriminals are motivated by illegal profit, for example, stealing money or sensitive data and exploiting malware for ransom payment. Hacktivists conduct similar attacks, but they are motivated by political, religious or social ideologies. The complexity of attacks is significant and successful attacks can completely disrupt systems leading to large-scale consequences. Attacks are not specifically targeted, but operate at large aiming to capture as many victims as possible, as was the case with recent attacks (*e.g.* WannaCry, Petya).

Modern threats can also be tailored and developed exclusively against a single organisation or country. Thanks to substantial financial support, a nation-state agent, can destabilize and disrupt targets.

Given this range of threats, a structured and complete vulnerability assessment procedure, which allows a timely response (*e.g.* patching), is vitally important. However, the patching procedure needs to be adequately conducted to ensure that it does not introduce new vulnerabilities.

All in all, to face modern threat agents, software systems must rely on a principled development process, on continuous monitoring for undetected vulnerabilities and on regular maintenance.

5 Method

The large variety of threats requires us to adopt a multi-faceted approach, bringing together safety and security guidelines into a structured group of expectations. We present here the rationale behind our approach and the organisation of our outcomes.

5.1 Approach

Successful development of critical software relies on a combination of factors, including processes, tools and skills. When identifying factors, we have taken inspiration from a number of sources, including: the "4+1" software safety assurance principles (Hawkins, 2013); software considerations in airborne systems and certification (RTCA, 2011); design assurance guidance for airborne electronic hardware (RTCA, 2000); data safety guidance (DSIWG, 2018); functional safety of programmable electronic safety related systems (BSI, 2010); technical risk assessment and risk treatment (HMG, 2012); and a framework for improving critical infrastructure cyber security (NIST, 2018).

Building from those sources and drawing on our own experience we have identified things that we expect from every development of critical software. When eliciting these expectations our focus was tightly restricted to things within the remit of a software development team (rather than, for example, a systems team or a corporate entity). This focus was adopted because of our experience of conducting, assessing and auditing software development. More specifically, our expectations could potentially form the basis of a combined safety and security assessment of software development activities.

It is important to acknowledge our focus means many things that are important to safety (*e.g.* system-level hazard assessment) are omitted. Likewise, many things that are important to security are also omitted (*e.g.* building access controls).

The desire to support a range of approaches, along with the immaturity of the expectations, meant we chose not to define a large number of criticality levels. Instead, we simply identify common expectations and, where necessary, supplement these with additional pieces that apply to more critical CSCIs; that is, cases where a greater level of assurance is required.

Our aim was to produce a set of sufficiently-generic expectations that would apply to every development, yet which contained sufficient detail to allow them to be of practical utility, whilst also allowing sufficient flexibility for different software development philosophies (*e.g.* waterfall, iterative) and approaches. Flexibility is also important, as we anticipate these expectations being refined as they are used: whilst we believe many of the core themes will remain constant, the details are likely to change.

In summary, we were targeting a region that was more detailed than the "4+1" software safety principles, but was less rigid than, say, the 71 objectives of DO-178C (RTCA, 2011).

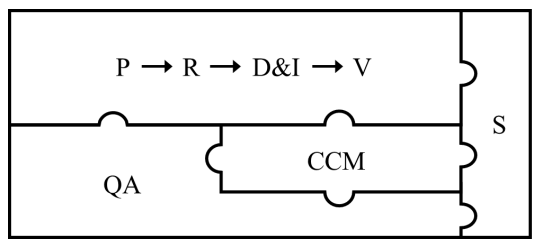
5.2 Organisation

Given our approach, we have found it convenient to organise the expectations into three broad groups:

- Those that are *underpinning*, in the sense that they apply throughout a software development process. This includes:
  - Quality Assurance (QA);
  - Change and Configuration Management (CCM).
- Those that apply to particular *phases* of software development. This includes:
  - Planning (P);
  - Requirements (R);
  - Design and Implementation (D&I);
  - Verification (V).

  Note that our use of *phases* does not imply use of a particular software development philosophy; mapping our phases on to a particular development philosophy should be straightforward.
- Those that highlight *specific items* (S) that warrant particular attention, including: use of algorithmic data; management of subcontractors; tooling; and staffing. Note that this list of specific items results from practical experience and aims at tackling a broad part of complementary development activities. However, there is no guarantee that all relevant items have been captured.

These three groups are not independent, instead they are tightly coupled. Intuitively, expectations associated with phases build upon those associated with underpinning processes, whilst specific items bring in items that are neither underpinning nor restricted to a single phase. This coupled relationship between the headings in these three groups is illustrated in Figure 1.



P - Planning; R - Requirements; D&I - Design and Implementation; V - Verification; S - Specific Items; CCM - Change and Configuration Management; QA - Quality Assurance

Fig. 1. Relationship between headings used to organise expectations

Similarly, there can be overlap between the individual expectations. For example, the need for a QA plan can be considered as both a QA expectation and a planning (P) expectation. From our perspective, relationships between expectations provide added strength, as they make omissions less likely.

6 Expectations

This section describes our expectations, providing practical information to help their interpretation and application. All expectations are intended to apply to large critical software development projects, regardless of the motivating reason for the criticality.

6.1 Underpinning

Quality Assurance (QA) – *Planned and monitored review controls assuring the quality of the development*. For a critical system it is important that QA activities are independent of the engineering activities. These activities will support the development of a robust product in a number of ways, including, providing independent evidence that processes have been followed and supporting continuous process improvement.

> **Expectation QA1 – QA Process Plan:** The developer is expected to implement an independent, planned QA process that provides evidence that defined processes have been followed. For CSCIs that require a greater level of assurance, an additional pre-release review, which provides independent evidence that all processes are complete, would be expected.
>
> **Expectation QA2 – Independent QA Monitoring:** The QA plan is expected to establish an independent group of people who will monitor the software development activity.

Change and Configuration Management (CCM) – *Tracking the evolution of software-related artefacts.* In any critical system, CCM establishes and maintains a consistent record. This means, for example, any item released to operational use can be faithfully recreated. CCM is also required to prevent "enthusiastic" developers making unilateral changes.

> **Expectation CCM1 – CCM Plan:** The developer is expected to design a CCM process, which includes a problem reporting process. The plan must be based on defined configured items (*e.g.* planning documents; design artefacts; code; verification results; software development environment). For CSCIs that require a higher level of assurance, the plan would be expected to establish explicit baselines and traceability.

Changes are an intrinsic feature of software development. Typically, there are four reasons for a software change: (i) *adaptive changes*, which respond to new environments or standards; (ii) *corrective changes*, which address defects; (iii) *perfective changes*, which relate to new requirements; (iv) *preventive changes*, which ease future maintenance (Williams and Carver, 2010).

A key aspect of the modern software environment is the discovery of vulnerabilities. For example, zero-day vulnerabilities (*i.e.* those whose existence was not known publicly and whose nature is significantly different from known cases) would motivate corrective changes. In order to be confident that critical software will continue to perform throughout its life, there needs to be a process that scans for identified vulnerabilities and implements appropriate corrective action.

Expectation CCM2 – CCM Vulnerability Monitoring: The developer and independent reviewers are expected to continuously monitor for potential vulnerabilities. Should any be identified, appropriate changes should be implemented.

Within the safety-critical software domain the need to maintain software artefacts for a prolonged period is well understood. Extending our considerations to include security means we also need to consider how artefacts are destroyed. This is especially important for artefacts that are destroyed during the system's life.

Expectation CCM3 – CCM Data Management: Appropriate processes for the secure long-term archive, retrieval, release and secure destruction of CSCI-related information should be implemented.

6.2 Phase-Based

Planning (P) – *Definition and scheduling of development tasks.* Developing critical software is a complex, costly process, which needs to be appropriately planned and managed. Plans need to be sufficiently detailed to ensure that everyone understands what is expected of them (and when), but not so detailed as to constrain the application of sound engineering judgement.

A variety of supporting standards would also be expected. In addition, the documents would be expected to be made available for review by the customer or, in some cases, by other stakeholders (*e.g.* regulatory authorities).

Expectation P1 – Planning Documents: The developer's software development planning documents would be expected to cover: software development; verification; CCM (see also CCM1); and QA (see also QA1).

Expectation P2 – Development Guidelines: The developer is expected to define guidelines that support all phases of the software development process. To facilitate QA, methods of checking standards would also be expected. A minimum would be a coding standard (used in the implementation phase). For CSCIs that require more assurance, requirements and design standards would also be expected.

Expectation P3 - Customer Review: The developer is expected to make all relevant software development artefacts available for review by the customer (or their representative).

The information contained in software development artefacts is likely to be of value to adversaries who seek to deliberately undermine software behaviour. Some artefacts are likely to be more valuable than others. For example, a QA plan may reveal little about the software in question, whereas a detailed design document is likely to contain many useful details. This information needs to be appropriately protected, but in a manner that does not unduly prevent engineers from doing their jobs. The way this balance is to be achieved is an important topic, which should be covered in the planning documents.

> **Expectation P4 – Controlling Development Process Data:** The planning documents would be expected to consider the value of the information contained in all artefacts produced by the software development process. According to different levels of confidentiality, it is expected that access to and sharing of this information is controlled.

In many cases, critical software developments occur within legal and / or regulatory frameworks. Consequently, planning documents would be expected to support engagement with relevant authorities.

> **Expectation P5 – Legal and Regulatory Compliance:** The planning documents would be expected to ensure, via the production of auditable evidence, that legal and regulatory requirements (*e.g.* relating to cryptographic controls) are satisfied. The planning documents are also expected to highlight any novel or contentious areas that may make it challenging to produce suitable assurance evidence, so these areas can be discussed with relevant regulatory authorities.

Requirements (R) – *Definition of software functionality and criticality level***.** Requirements will be allocated to software from the system engineering process (or as part of contracts). These requirements must be consistent, implementable and verifiable. They should also define the level of criticality associated with each software function. These criticality levels should be established using a rigorous, auditable and repeatable process, which considers mission impacts, security impacts and safety impacts.

> **Expectation R1 – System-level Requirements:** The developer is expected to ensure that the initial system-level (or contract-level) requirements are: sufficient; consistent; and verifiable.

> **Expectation R2 – Requirement Traceability:** The developer is expected to develop software requirements that are traceable to system-level (or contract-level) requirements. Traceability is also expected from the software requirements to verification tests. For CSCIs that require a higher level of assurance, the software requirements would be expected to be split into high and low-level requirements, with traceability extended across these levels.

> **Expectation R3 – Software Requirements:** The developer is expected to demonstrate that the software requirements exhibit certain characteristics, for example they should be: accurate; sufficient; consistent; verifiable. If

criticality requirements conflict, the developer is expected to clearly identify these so an informed decision on priority can be made.

Design and Implementation (D&I) – *Defining the software architecture and developing the corresponding code.* The development process is a potentially complex activity, which should be informed via discussions with the system team. This activity refines the requirements through a number of levels to arrive at a design, an architecture and low-level requirements from which the software can be successfully implemented. Traceability is expected to be maintained through this process (and beyond).

Expectation D&I1 – Design Activity: The developer would be expected to complete a design activity that results in an artefact that can be directly coded against. Artefacts from the design process would be expected to be accurate, consistent and verifiable. For CSCIs that require more assurance the design artefacts would be expected to include explicit descriptions of the architecture and low-level requirements.

Expectation D&I2 – Extended Traceability: The developer is expected to maintain traceability from system level requirements through software requirements, design, implementation and verification.

The design is expected to include features that provide security protection. This includes protection to maintain confidentiality, integrity and availability of data, as well as protection to prevent one CSCI being able to affect the behaviour of another in a way that was not planned and designed for. This latter consideration includes protection against malformed, or otherwise inappropriate, input data, as well as protection of a CSCI's memory space.

Expectation D&I3 – Design Security Protection: The design would be expected to provide provision for sufficient protection of data, both at rest and during transit. The potential for data leakage would be expected to be considered, as would the need to validate the integrity of information that is received.

Expectation D&I4 – Partitioning: The design would be expected to include appropriate partitioning, taking into account criticality considerations. CSCIs that require more assurance would be expected to have specific protection against random failures, chance events and deliberate hostile actions.

Given the criticality of the system, it is expected that the design phase should include assessment of algorithms. This is because algorithms, especially numerical ones, have properties that may be difficult to exhaustively examine during testing. For example, aerodynamic equations can become unstable as airspeed approaches the sound barrier. This instability may not be immediately obvious from test results; aircraft have been released to front line service without it being detected (Alberico, 1999). Consequently, we would expect algorithms to be reviewed by knowledgeable parties, including experts in the algorithm's domain and experts in

issues associated with representations of real numbers in computational hardware (Goldberg, 1991).

> **Expectation D&I5 – Algorithm Accuracy:** The developer would be expected to confirm the accuracy of the algorithms that are used. This may be achieved through, for example, knowledgeable review or simulation or prototyping.

Verification (V) – *Confirming software functionality and robustness***.** The evidence necessary to support the assurance of critical software comes from activities throughout its development. In particular, this evidence is not simply restricted to results from dynamically executing the software on test cases. For example, we would expect some form of code review.

> **Expectation V1 – Code Review:** The developer would be expected to conduct some form of code review. This may be a formal analysis or peer review, targeted at key parts of the code and may be tool assited. For CSCIs that require more assurance, this could include static analysis to confirm compliance with a coding standard; it may include formally-documented peer review of all code (or artefacts from which code is automatically generated).

Of course, testing remains an important activity. We would expect requirements-based tests (with, as per D&I2, traceability maintained from system-level requirements to test cases). These tests should cover cases of expected behaviour; that is normal test cases. They should also cover cases where the CSCI is subjected to unexpected inputs and situations (*e.g.* abnormal initialisation); that is, robustness test cases. Robustness cases would also be expected to include examples where an adversary deliberately tries to adversely affect software behaviour.

> **Expectation V2 – Requirement-based Test Cases:** The developer would be expected to develop requirements-based tests. Normal test cases and robustness test cases would be expected, and every requirement would be expected to be covered by at least one test. Evidence would be expected to demonstrate the test procedures are correct and the test results are accurate. For CSCIs that require higher assurance a particular level of structural coverage (*e.g.*, statement coverage, branch coverage) would be expected; the level of coverage is expected to be commensurate with the CSCI's criticality.

> **Expectation V3 – Security-based Test Cases:** The developer would be expected to develop tests based on potential activities that may be undertaken by an adversary. These tests are expected to strengthen classical robustness testing, for example, tests of partitioning would be expected to explicitly cover cases where a threat is actively trying to defeat the partitioning.

> **Expectation V4 – Test Review:** The developer is expected to instigate a review of the test procedures. Independent running or observation of tests

would be expected to be formally-documented procedures. Tests of parameter data files would be expected to explicitly cover cases where the data is incomplete or incorrect.

6.3 Specific Items

Modern software systems have many items that may be configurable via data, either once at system load and start up, or multiple times depending on the phase of the application. The key concepts of managing this type of data are that: any allowable data set will not cause an error; only allowable data sets are used; and data sets are treated with a rigor commensurate with the criticality level of the CSCIs they may influence. There is also a need to demonstrate that data artefacts exhibit the required properties (*e.g.* timeliness, accuracy) (DSWG, 2018).

Expectation S1 – Algorithmic Data: The developer is expected to explicitly identify any algorithmic data being used and to ensure it is controlled and managed with the appropriate rigor. For example, the developer would be expected to consider whether specific data artefacts need to exhibit particular properties and, if so, to ensure these properties are maintained.

Software systems often include components produced by multiple suppliers. Whilst the integrator remains responsible to the customer, it is important that any subcontractor is appropriately chosen and suitably managed (including ensuring that all required evidence is available).

Expectation S2 – Subcontractor Management: The developer is expected to have a documented process, including criteria, for the selection of subcontractors. The contractor is expected to monitor subcontractors; this monitoring includes collection of evidence so that a coherent, comprehensive body of assurance evidence is available for the CSCI.

Although the integrator is responsible for subcontractor management and performance, the integrated nature of the overall software means that the final customer may need to obtain evidence from subcontractors. The right to do this has to be included in contracts.

Expectation S3 – Customer Audit: The rights of the customer to review planning documentation and to audit their implementation, including the resulting artefacts, is expected to be extended to all subcontractors.

Humans have been using tools to make their job easier for millions of years and software engineers are continually developing tools to make the engineering process more efficient.

Expectation S4 – Tools: The developer is expected to ensure that any tools used in the development of a CSCI are of a suitable quality and, if

appropriate, qualified to an appropriate level. A greater amount of evidence is expected for tools that can introduce errors into the software than for tools that may only fail to detect errors.

In any project having staff with appropriate skills (*e.g.* both domain knowledge and software engineering skills), and supporting those still acquiring these skills, is an important aspect of project management. This becomes more important on projects with long development or long in-service periods, as staff changes are more likely.

Expectation S5 – Staff: The developer is expected to have a process to ensure staff are appropriate for the task, this can be achieved via assessment of skills and experience, and to support relevant training, without jeopardizing the quality of the software. The competencies considered should include: software engineering; information security; safety and domain (mission) knowledge.

7 Conclusions

We have developed and described a set of expectations that should be met for any development of critical software. These are summarised below.

- Quality Assurance (QA):
  - QA1: QA Process Plan;
  - QA2: Independent QA Monitoring.
- Change and Configuration Management (CCM):
  - CCM1: CCM Plan;
  - CCM2: CCM Vulnerability Monitoring;
  - CCM3: CCM Data Management.
- Planning (P):
  - P1: Planning Documents;
  - P2: Development Guidelines;
  - P3: Customer Review;
  - P4: Controlling Development Process Data;
  - P5: Legal and Regulatory Compliance.
- Requirements (R):
  - R1: System-level Requirements;
  - R2: Requirement Traceability;
  - R3: Software Requirements.

- Design and Implementation (D&I):
  - D&I1: Design Activity;
  - D&I2: Extended Traceability;
  - D&13: Design Security Protection;
  - D&I4: Partitioning;
  - D&I5: Algorithm Accuracy.
- Verification (V):
  - V1: Code Review;
  - V2: Requirement-based Test Cases;
  - V3: Security-based Test Cases;
  - V4: Test Review.
- Specific Items (S):
  - S1: Algorithmic Data.
  - S2: Subcontractor Management;
  - S3: Customer Audit.
  - S4: Tools.
  - S5: Staff.

From the perspective of a safety specialist, it may be tempting to consider the above list as being just a minor extension on existing practice. For example, CCM2, which relates to vulnerability monitoring, is readily identifiable as an extension introduced for security reasons. Conversely, many of the other items will be familiar to those with experience in safety-critical software.

However, to view the list as, in some way, "safety-critical software plus a bit" misses the point. In particular, our aim was to create an integrated list of expectations that addressed safety, security and mission criticality. As such, an individual expectation can address issues from multiple perspectives. An obvious example is D&I4, which relates to partitioning; this is common practice in safety-critical software, but is also of great value from a security perspective.

Whilst there may be much debate about the specific expectations we have chosen, we hope the need for a holistic software development process that considers all forms of criticality is clear. We also hope our work has made a small contribution to that end.

Acknowledgments BS and RA would like to thank Andy Adams of the Defence Airworthiness Team, Defence Equipment and Support (DE&S), for his support and encouragement. RA, AM and VS would like to thank the Cyber Security Academy, Southampton University, for many stimulating conversations, which have informed this work. In addition, we note the paper benefitted significantly from comments received during the review process for the 2019 Safety-critical Systems Symposium.

References

Alberico D, et al. (1999) Software System Safety Handbook. Joint Software System Safety Committee.
Ash D (2007) Lessons from Epinal. Clinical Oncology 19.8 614-615.
BSI (2010) Functional safety of electrical/electronic/programmable electronic safety related systems. BS EN 61508:2010.
DSIWG (2018) Data Safety Guidance. Version 3, SCSC-127C.
Goldberg D (1991) What every computer scientist should know about floating-point arithmetic. ACM Computing Surveys (CSUR) 23.1 5-48.
Hawkins R, Habli I, Kelly T (2013) The principles of software safety assurance. 31st International System Safety Conference, Boston, Massachusetts USA.
Howard G, Butler M, Colley J, Sassone V (2017) Formal analysis of safety and security requirements of critical systems supported by an extended STPA methodology. At 2nd Workshop on Safety & Security Assurance, 2017.
HMG (2012) IA Standard Numbers 1 & 2 - Supplement Technical Risk Assessment and Risk Treatment.
Ishimatsu T, et al.(2014) Hazard analysis of complex spacecraft using systems-theoretic process analysis. Journal of Spacecraft and Rockets 51.2 509-522.
ISO (2017) Systems and software engineering - Vocabulary. BS ISO/IEC/IEEE 24765:2017.
Kali Linux (2018) Penetration Testing and Ethical Hacking Linux Distribution. Available from https://www.kali.org.
Matthews L (2017) Criminals hacked a fish tank to steal data from a casino. Forbes. https://www.forbes.com/sites/leemathews/2017/07/27/criminals-hacked-a-fish-tank-to-steal-data-from-a-casino/#ec931b632b96.
NIST (2018) Framework for improving critical infrastructure cybersecurity, Version 1.1.
RTCA (2000) Design assurance guidance for airborne electronic hardware. DO-254.
RTCA (2011) Software considerations in airborne systems and equipment certification. DO-178C.
SAE International (2010) Guidelines for Development of Civil Aircraft and Systems. Aerospace Recommended Practice (ARP) 4754A.
Williams B J and Carver J C. (2010) Characterizing software architecture changes: a systematic review. Information and Software Technology 52.1 (2010): 31-51.